AN INVESTIGATION OF THE PARAMETERS FOR REGISTER ALLOCATION DURING COMPILATION

Van Douglas Underwood

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN INVESTIGATION OF THE PARAMETERS FOR REGISTER ALLOCATION DURING COMPILATION

by

Van Douglas Underwood

June 1974

Thesis Advisor: G. A. Kildall

Approved for public release; distribution unlimited.

An Investigation of the Parameters for Register Allocation During Compilation

by

Van Douglas Underwood
Lieutenant, United States Navy
B.S., Iowa State University, 1968

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the
NAVAL POSTGRADUATE SCHOOL
June 1974

ABSTRACT

Two established methods of code improvement, Day [4] and Kildall [7], are reviewed. The problems of optimal register allocation are discussed. A method is presented using Kildall's [7] optimization algorithm for specifying the active data items in a program. Demonstraticn of particular problems with register allocation are presented. Topics for further consideration in a complete soluticn are discussed.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

The author wishes to extend personal and heartfelt thanks to Gary A. Kildall whose efforts as a professor provided the interest in the subject area, whose efforts as an academician provide the structure for many of the ideas presented, and whose patience and time spent in review helped keep fantasies founded in reality.

Also the author wishes to thank Ensign Gary Raetz who gave much time as a student and laboratory partner and later as a second reader to many efforts which helped both directly and indirectly in the development of this thesis. In conjunction with Gary Raetz's time, thanks are also extended to his wife Mary for understanding his absences from her.

II. BACKGROUND

The advent of higher order languages began the era of compilers and subsequently optimizers. The problem with compilers in general is that more efficient code can often be written by an assembly level programmer. Efficiency in this case is measured in the execution time for the program, the amount of memory required to store the program, or both. The desire for improved code lead to the development of the science of code optimization. The field of optimization has been explored [1,3,6,7,10] and several basic theories have evolved.

One important aspect of optimization is register allocation. The register allocation problem is the problem of assigning data items to registers so that the resulting code is efficient. The extension of the problem is to coordinate register assignments made on different branches and to determine data item replacement. Techniques will be presented for examining these two problems.

The discussions which follow will relate to a class of computers which are characterized by having a set of general purpose registers. The methods discussed would not apply to stack machines, for example.

Data item allocation will be to one of a set of general purpose registers. The methods will then apply to the allocation problem by data item type. That is to say, the methods may be applied to allocating floating point data items to a set of floating point registers and integer data items to a set of integer registers as long as the data item type can be identified by the compiler.

Several authors have addressed the problem of optimal register allocation, either directly or indirectly. It will be the purpose of this tretise to examine two of these techniques [Day 4,Kildall 7] with the purpose of determining

the optimizing information which is necessary for optimal allocation.

A. OPTIMAL REGISTER ALLOCATION, DAY [4]

One solution to optimal register allocation has been presented by Day [4]. Day's method is based primarily on the concepts of data item interference and profit.

Day defines a data item as a constant or a data name. "A data item is _defined_ when statement execution causes a new value to become associated with the data item." Constants are defined by their representation and their values are not normally changed. "A data item is _referred to_ when the current value of the data item is required for correct statement execution." Having the current value required for correct statement execution implies that the value is at least temporarily in a register. "A data item is _active at a point_ in" a region "if it may be referenced subsequent to that point." Day defines a region to be a strongly connected subgraph of the program when represented as a directed graph. A strongly connected subgraph, by definition, means that any node in the subgraph can be reached from any other node in the subgraph. Because of this characteristic of strong connectivity when combined with the definition of an _active_ data item, a data item is necessarily active over an entire region if it is active at any point in the region. Strongly connected regions intuitively correspond to nested loop structures in the source program. Day states that "two data items interfere in" a region "if they are both active at a point in" the region. Extension of the concept of active data items implies that any data items active in a region must necessarily interfere with any other active data item in the region at all points in the region. Since a region is strongly connected, all points in the region are necessarily subsequent to every other point in the region.

The principal characteristic of interfering data items is that if they are allocated to the same register, at some point they will both be active, by the definition of iterference, and may not be allocated to the same register at that point. Likewise, the characteristic of non-interfering data items is that at no point are any two non-interfering data items active.

The concept of profit is a numerical "representation of the improvement in program execution that may occur if the data item is globally assigned to a register being processed." The comparative values of the profits are the deciding factors in making the assignment of a data item to a register. "The values assigned to the profit equation constants determine whether the profit represents a projected improvement in program size or execution time." Day assumes "the profit of a particular global assignment to be the sum of those data items therein assigned to registers." In terms of the analysis, the method is to maximize over all possible assignments to identify the assignment with the largest profit value.

A basic block is an ordered set of statements

$$\{S1,S2,S3,...,Sk\}$$

which is entered only through S1 and branched from only at Sk and where Si is executed before Si+1. Day uses this definition in the discussion of both local and global assignment. "Local and global assignment differ in the extent of the program over which the assignment of data items to registers is effective: local assignment occurs within a basic block, while global assignment occurs within a region."

The desirability of global assignment stems from the weakness of the more easily conducted local assignment. Day states that one "weakness in local assignment involves the disposition of data items that are defined or referred to in a block and are active on entry to or exit from the block.

Local assignment cannot usually retain assignment history across block boundaries, and so the values of active data items must be moved to main storage for interblock transfers of control."

Day introduces three types of allocation: one-one, many-one, and many-few. "A one-one assignment defines a one-to-one correspondence between" the data items and the registers. A weakness in global one-one assignment is that it is usually incapable of assigning more than one data item to a register in a region. Day's approach to the solution of the problem "is to consider a set of data items for assignment to a register if no two data items in the set interfere at any point in the region." Day's global many-few assignment method has this characteristic. Many-few assignment is a single valued mapping of a subset of the data items in a program onto a set of registers where the number of data items competing for assignment is greater than the number of available registers.

Day presents a solution method for the global many-few problem utilizing matrix construction and multiplication to implement the interference characteristics of the data items.

B. GLOBAL EXPRESSION OPTIMIZATION, KILDALL [7]

Kildall conducts an analysis of program structure in order to produce optimized object code. Kildall utilizes a directed graph to represent the program flow, along with an "optimizing pool," an "optimizing function," and a "meet operation" to conduct his analyses.

An optimizing pool is associated with each node in the graph. The nature of the pool is an arbitrary set which describes the optimizing information associated with a particular node in terms of the analysis being conducted.

An optimizing function maps an "input pool" and the optimizing pool of a node to a new "output" pool. In every

instance, the input pool for a node is derived from the output pools for the node's immediate predecessors. The output pool of a node contributes to the input pool of the node's immediate successor(s).

The meet operation is defined to handle the problem of combining two or more input pools at a point where two or more program flows join, and varies for differing types of analysis. The meet operations defined are binary, associative, and commutative. The meet operation is a mapping of the set of all optimizing pools onto itself. This can be represented as:

$$\underline{P} X \underline{P} \rightarrow \underline{P}$$

(where $\underline{P}$ is the set of all optimizing pools).

Kildall defines several types of analysis. Two of his analyses are of primary interest and will be reviewed below. The two are common subexpression elimination and live variable analysis.

For common subexpression analysis, the pool of computed expressions is partitioned into equivalence classes whose members are known to have identical values. The optimizing function for common subexpression analysis manipulates the equivalence classes of the partition. "Two expressions are placed into the same class of the partition if they are known to have equivalent values." The meet operation for common subexpression analysis is intersection by equivalence classes.

For live variable analysis a reversed program flow graph is used for the analysis. At any point, the pool associated with a node is the set of data items which may be referenced subsequent (in the forward direction) to the node. The optimizing function for live variable analysis has two distinct characteristics. These are:

"1. If the expression at node N involves an assignment to a variable, let d be the destination of the assignment; set $P \leftarrow P - \{e | d$ is a subexpression in $e\}$ (d and all expressions containing d become dead

expressions)" (e is the set of all partial computations at the current node.)

"2. Consider each partial computation e at node N. Set P←P {e} The value of the optimizing function is altered to the value of P."

The meet operation for live variable analysis is set union.

For completeness, Kildall's flow analysis algorithm is presented below. The following notation will be used in the presentation: $\underline{P}$ is the set of all possible optimizing pools. $\underline{E}$ is an entry pool set. $\underline{1}$ is the unit element for the analysis being conducted.

```
A1[initialize]    L ← E

A2[terminate?]    If L=∅ then HALT

A3[select node]   Let L'∈L,L'=(N,Pi) for some N∈N and
                  Pi∈P, L←L-{L'}

A4[traverse?]     Let Pn be the current approximate pool
                  of optimizing information associated
                  with the node N (initially Pn=1).  If
                  Pn≤Pi Go To step A2.

A5[set pool]      Pn←Pn∧Pi,L←LU{N',f(N,Pn))|N'∈I(N)}

A6[loop]          Go To step A2.
```

Examples of optimizing pools, an optimizing function, and a meet operation are presented in section V. The term global will be used henceforth to refer to an entire program and not just a region.

By utilizing Kildall's methods, the data item concept can be expanded to include expressions. This extension is desirable because a repeated expression would have to be recomputed if allocation were only to variables and

constants. Kildall [7] presents a data structure which may be used for manipulating the data items under the expanded definition.

III. THE CONCEPT OF PROFIT

The concept of profit is essential to register allocation. Day's definition of profit is a linear combination of the number of definitions of and references to a data item in a region.

The expanded definition of data item may mean that the number of references may not accurately and completely reflect the value of a data item (an expression, for example) in a register. It will not be the purpose of this paper to specify an explicit profit function. However, the contributing factors of the profit function under the expanded data item definition will be discussed below.

In general, it is assumed that the profit should reflect a measurable quantity. If the optimization is to be toward program size, the profit function should be a measure of the relative number of instructions required to execute the resulting code. If the optimization is to be toward program run time, the profit function should be a measure of the execution time of the resulting instructions. The two concepts, of course, are often closely related.

The profit should increase with the number of references to a data item over an active region. If a high profit results from this factor, it would imply a decrease in the number of load operations (and combining operations in the case of expressions). This factor would then imply a decrease in program size and, depending on the operation times, often leads to savings in run time.

The profit should increase with the number of instructions necessary to replace the data item in a register, due to the fact that the data item concept is extended to include arbitrary expressions. This factor is called "complexity," since the profit is related to the complexity of the data item. To reduce run time, profit

should assign different values to the operations as to execution time. The slower execution times of a particular operation would lead to a _higher_ profit since it would require more time to replace the data item in a register. Exponentiation would be weighted more heavily than addition, for example.

The profit should decrease with increased distance to the next reference, thus preventing highly complex data items from holding a register over long program flows without reference.

The profit should be adjusted with program flow information, when available. Logically, a data item on a highly executed branch would have higher value in a register than a similar data item on a seldom executed branch.

IV. ACTIVE DATA ITEM EXTENSION

One of the primary attributes of Day's analysis deals with the concept of interfering data items. A problem with Day's definition of an active data item when combined with his definiticn of a region was mentioned above. It becomes desirable to make a new definition of an active data item to correct that problem. Intuitively, a data item is active between a definition of the data item and the last reference to the value of the data item which was thereby defined. In terms of register allocation, this definition may be stated as: a <u>data item is active</u> at all points where the value of the data item must exist or have existed in a register for proper statement execution and remains active to the last reference to the data item for which the value which existed in a register would yield correct execution. Infcrmally, in terms of registers, a variable becomes active when the associated value exists in a register and remains active over the range to the last point at which it is referenced prior to redefinition or program termination. In other words, it is the range over which a data item maintains the value which was at one time loaded into a register.

To an extent, live variable analysis corresponds tc this definition. Variables are included as being live over the range from which they are assigned a value by an executable statement to the point of their last reference prior to redefinition (by an executable statement) or program termination. Live variable analysis departs from the definition given for active data items in two ways.

The first departure of live variables from active variables comes from the case of data items which are implicitly defined. Implicit definition may be made by the representation (in the case of constants), by compile-time

assignments (e.g., the FORTRAN "DATA" statement), or by default memory initialization. Implicitly defined data items are evaluated as live from program entry to the last reference to the data item (with possible non-active sections interspersed). For implicitly defined data items, however, the value associated with the data item does not exist in a register until the first reference. Implicitly defined data items are, therefore, active from their first reference to their last reference or redefinition.

The second departure of live variables from active variables may occur from a READ-type statement. Depending on the machine configuration and the data manipulation for a READ-type statement, the data item read may or may not have existed in a usable form in a register. Live variables begin a live segment with the definition by a READ-type statement. Depending on the data manipulation, a READ statement may or may not originate an active program segment for that data item.

V. REFERENCED DATA ITEM ANALYSIS

In order to analyze data item interference based on the revised definition of active data items, referenced data item analysis is introduced. The purpose of referenced data item analysis is to provide a method which determines data items possessing the characteristics of active data items not possessed by live variables so that the active data items may be determined. In particular, referenced data item analysis produces sets of data items which have previously appeared in a register.

The optimization pool for referenced data item analysis is the set of all data items which have been referenced previous to the current point in the program flow. The optimizing function performs a union of all data items in the expression at the current node with the input set of referenced data items. Thus for an expression R=A+B at a node N with an input pool of {X,Y,Z}

$$F(N,\{X,Y,Z\})=\{X,Y,Z\} \cup \{R,A,B\}=\{R,A,B,X,Y,Z\}$$

where F(N,Pn) is the optimizing function operating on node N and the corresponding input pool Pn.

Active data items are not dependent upon the program branch structure. An active data item is active from a first reference (loaded into a register) to the final reference, with possible inactive segments interspersed. The meet operation is, therefore, set union.

As discussed above, the inclusion of a variable in the referenced data pools which participate in a READ-type statement will be dependent upon the machine for which the output code is intended.

As discussed in section IV, live variables have the characteristics of active variables with the exception that their point of entry into the set of active data items may

overextend the definition point. Referenced data items have the characteristics of active data items except that the data items may (and in general will) extend past the last reference. Intersection of pools for these two analyses at every point in the program flow, then, will produce sets of data items which are active at that point. It should be noted that a single forward pass is insufficient for active variable analysis because on a forward pass the current reference to a variable is not known to be the last. Similarly, a single reverse pass is insufficient because the current reference is not known to be the first reference.

VI. A PROBLEM WITH ALLOCATION

The problem of register allocation is now considered. There are certain necessary requirements which are implicit because of the interference characteristics of the variables. The nature of these necessary requirements is that no two mutually active variables may be assigned to the same register. There are some desired characteristics which are derived from the branching structure of the program. The nature of the desired characteristics is that variables active on several branches be in the same register at the point where the branches join. The necessary and the desired characteristics will now be discussed.

At any point in the program, only one data item may be allocated to a register. Day [4] allocates a set of non-interfering data items to a register. The characteristics of non-interfering data items, however, imply that only one data item at any given point will have value in the register (be active at that point). Therefore, although Day allocates a set of data items, at any point only one member of that set will be in a register.

Further, all active data items have value in a register. If the number of active data items is greater than the number of registers, then a selection must be made of the data items to be allocated. Profit is the measurement used to make the selection. The selection may be made by reducing the set of active data items at each node to the M most profitable data items, where M is the number of registers. The necessary requirements will be derived from these reduced active data item pools. That is, no two members of any reduced active pool at any point may be allocated to the same register at that point.

It is desirable that data items allocated on different branches and are active at a point where the branches join,

be allocated to the same register on each branch. By meeting this desired characteristic, the register will hold the correct value of the data item regardless of the branch taken to reach the point where they join. The desired characteristics apply only in the event the data item is active on all branches which join at a point. If the data item was not active just prior to the join point, the contents of a register would be dependent upon the branch taken at run time.

Figure I is now presented to illustrate the desired and the necessary characteristics of a program segment. There are three variables in the example, X,Y and U. For purposes of the example, Ra(d) and Rb(d) will represent the register allocated to the data item represented by d on branch a and b respectively. R1 and R2 will represent the actual registers in the two register machine.

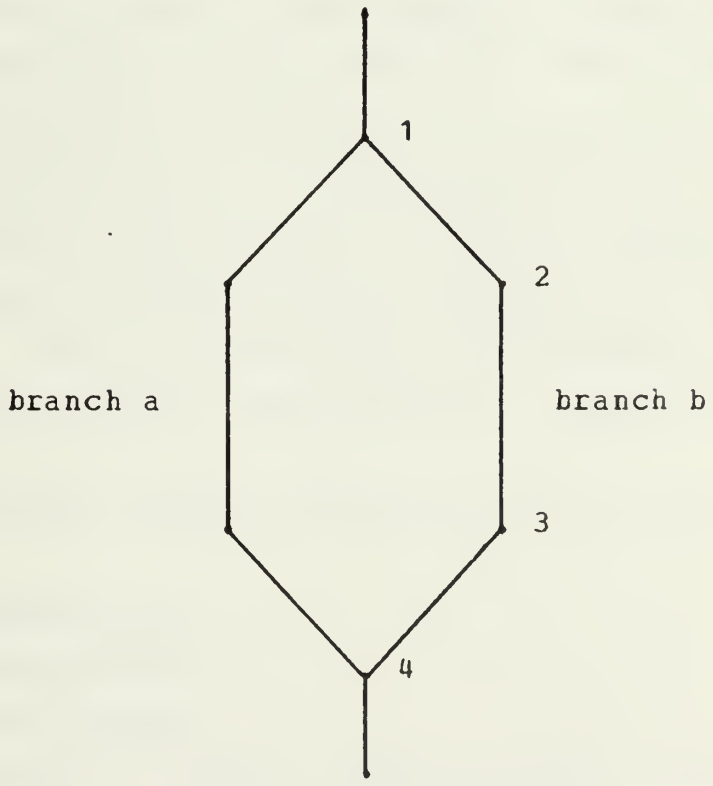


FIGURE I.

X and U are active prior to point 1 and over all of branch a. X is active on branch b to point 2 and from point 3 to point 4. Y is active on branch b from point 1 to point 3. U is active on branch b from point 2 to point 4.

The necessary requirements in the example are shown belcw.

1 $Ra(X) \neq Ra(U)$

2 $Rb(X) \neq Rb(Y)$

3 $Rb(U) \neq Rb(Y)$

4 $Rb(U) \neq Rb(X)$

The desired characteristics in the example are:

1 $Ra(X) = Rb(X)$

2 $Ra(U) = Rb(U)$

Another set of constraints stems from the activity characteristics of the variables in conjunction with the necessary requirements. At point 1, for example, $Rb(X) \neq Rb(Y)$ and at point 2 $Rb(U) \neq Rb(Y)$. Since there are only two registers in the machine, these requirements combine tc imply that $Rb(X) = Rb(U)$.

More specifically, the register released by a data item when it becomes inactive is subsequently used by another data item when it becomes active. Thus the register used by the first item is "equated to" the register of the second data item. Due to the fact that the activity of the data items must nct conflict, this action is termed "complement equation."

The ccmplement equation characteristics cf the example are:

1 $Rb(U) = Rb(Y)$

2 $Rb(X) = Rb(U)$

3 $Rb(Y) = Rb(X)$

The complement equation characteristics, the necessary requirements, and the desired characteristics combine to reach a contradiciton. The contradiction may be represented by :

Ra(X)=R1	Arbitrary
Ra(U)=R2	Necessary Requirement 1
Rb(Y)=R2	Complement Equation 1
Rb(X)=R1	Necessary Requirement 2
Rb(U)=R1	Complement Equation 2
Rb(X)=R2	Necessary Requirement 4
Rb(X)=R1	Desired Characteristic 1

Note that the last two assignments are in conflict. Arbitrarily setting Ra(X)=R2 would lead to the same contradiction.

The effect of this contradiction is that, if an alteration is not made to at least one of the characteristics, a reference to X or U after point 4 would require reloading of the variable being referenced.

locally using globally derived information, using Kildall's techniques [7].

B. COMPLEMENT EQUATION ANALYSIS

The concept of complement equation was introduced in section VI. The projected purpose of complement equation analysis is to specify the complement equation constraints of a program. Complement equation, being based on the activity characteristics of the data items, should operate on the sets of active variables. By comparing the pools of active data items from node to node, the changes in the pools represent the registers of the data items which are complement equated.

C. DESIRABLY EQUAL REGISTERS

Desirably equal register assignments occur where two or more program flows join. The program characteristic which leads to desired register equation is as follows: registers of data items are desirably equal if they are active on two or more branches prior to a join point, and are active at the join point, Specification of this situation may be made by intersecting the active pools prior to the join point with the active pool at the join point. Data items which are in the intersection are then desirably in the same register on all branches.

D. REGISTER PRELOADING

Preloading is the process of loading a register prior to a join point. Preloading may be desirable in two situations.

The first situation arises from the expanded definition of data items to include expressions. The situations may be detected by comparing the complexity of the common

subexpression pools from pre-join nodes to the join node. If the complexity of the pool increases, the preloading may be profitable. For example, if the pool structure is

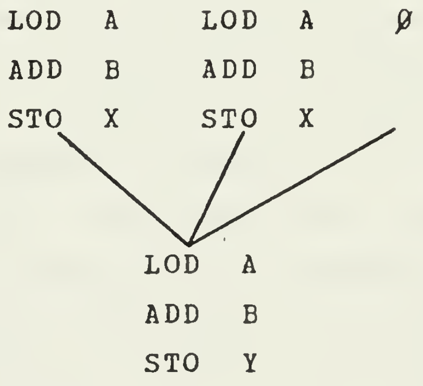
$$\begin{tikzcd}
\{X,A+B\} \arrow[dr, dash] & & \{Y,A+B\} \arrow[dl, dash] \\
& \{A+B\} &
\end{tikzcd}$$

it may be worthwhile forcing A+B to a register at the join point. To load A+B at the join point would require

```
LOD   A
ADD   B
```

However, prior to the join point, A+B may be loaded into a register by loading either X or Y requiring only one operation.

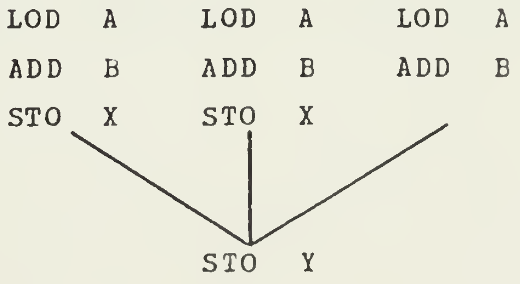
The second situation for which preloading may be of value occurs when several branches join and a data item is active on several, but not all of the branches joining at that point. By preloading the data item on the branches on which it is not active, the code on the branches may be fully utilized. Thus the code



```
LOD  A     LOD  A     Ø
ADD  B     ADD  B
STO  X     STO  X

      LOD  A
      ADD  B
      STO  Y
```

could be modified to



```
LOD  A     LOD  A     LOD  A
ADD  B     ADD  B     ADD  B
STO  X     STO  X

      STO  Y
```

in which case the correct value of A+B would be in a register and could be stored at the join point. The resulting code in this case would have the same program size, but would execute in a shorter time, especially if the right branch were seldom executed.

E. ALTERATION OF THE COMPLEMENT EQUATION CHARACTERISTICS

The complement equation characteristics are the constraints which must be relaxed. That is, given that a contradiction exists, the necessary characteristics cannot be changed, and while the desirably equal registers constraints may be altered, this could lead to excessive load-store operations. The opportunities for altering these charcteristics exist in three forms. If there is a node-to-node change in the active pools of two or more data items, the newly active data items may be loaded into any of the vacated registers. If there are fewer than M active data items at a node (where M is the number of registers), then the complement equation characteristics may be altered by performing a register-to-register move. The third alteration made be made at any point by storing the current register contents to a temporary location, performing a register-to-register movement, and loading the vacated register from the temporary location.

The purpose of altering the complement equation characteristics is to satisfy the desirably equal register constraints. When the alterations are made at a cost, such as storing, performing a register-to-register move, and loading, the cost would have to be balanced against the advantages gained by satisfying the desirably equal register characteristics.

X. CONCLUSIONS

Using Kildall's [7] algorithm, a method was presented for specifying the active data items. The concept of reducing the active data items at each node to the M most profitable (where M is the number of registers) was introduced. A description of the nature of the desired and necessary characteristics of allocation was presented.

The existence of the contradictions specified in section VI implies that there may not be a satisfactory solution to every register allocation problem. If there is no universally satisfactory solution, the problem then becomes a linear programming problem. As in Day's solution, the problem may be informally stated as:

MAXIMIZE: Profit
SUBJECT TO: 1. Necessary Requirements
2. (Desired Characteristics)'

where (Desired Characteristics)' may be a proper subset of the desired characteristics of the program. Profit in this case is the sum of the profits of the data items assigned to registers and the equating profits less the equating costs.

Maximizing the profit of the variables at each node will ensure that the profit associated with the data items is maximum. The natural extension would imply that maximizing the profit of equating at each step would also result in a maximum profit globally. This may not be true, however, since changes to the complement equation characteristics, when made, are effective over all nodes subsequent to the node at which the alteration is made. Thus an alteration to gain a desired register equating may require other alterations in the complement equation characteristics which will have an associated cost. None of these statements

have been formally specified, however, and remain as topics for further investigation. In the final analysis, it appears that the techniques discussed here must be applied somewhat heuristically in an attempt to obtain a "good" allocation. This allocation may be incrementally improved but, considering the current state of the theory, no absolute statements are possible at this time.

BIBLIOGRAPHY

1. Aho, A.,Sethi, R., and Ullman, J., A Formal Approach to Code Optimization, Proceedings of a symposium on compiler optimization, University of Illinois at Urbana-Champaign, July, 1970.

2. Busacker, Robert G. and Saaty, Thomas L., Finite Graphs and Networks: An Introduction With Applications, McGraw-Hill Inc., 1965.

3. Cocke, J. and Schwartz, J., Programming Languages And Their Compilers, Preliminary notes, Courant Institute of the Mathematical Sciences, New York University, 1970.

4. Day, W., Compiler Assignment of Data Items to Registers, IBM Systems Journal, 8, 4(1970), P281-317.

5. Horowitz, L.,Karp, R., Miller, R., and Winograd, S., Index Register Allocation, Journal of the ACM 13, 1(Jan. 1966), P43-61.

6. Kennedy, K., A Global Flow Analysis Algorithm, International Journal of Computer Mathematics, Section A, vol. 3, 1971, P5-15.

7. Kildall, G. A., Global Expression Optimization During Compilation, Technical Report number TR72-06-02, University of Washington Computer Science Group, University of Washington, Seattle, Washington, June, 1972.

8. Roon, J., A Direction-Independant Algorithm for Determining the Forward and Backward Compute Points for a Term or Subscript During Compilation, Computer Journal 9, 2(Aug. 1966), P157,160.

9. Schnieder, V., On the Number of Registers Needed to Evaluate Arithmetic Expressions, BIT 11(1971), P84-93.

10. Sethi, R. and Ullman, J., The Generation of Optimal Code For Arithmetic Expressions, Journal of the ACM 17, 4(Oct. 1970), P715-728.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
2.	Department Chairman, Code 72 Computer Science Group Naval Postgraduate School Monterey, California 93940	1
3.	Asst. Professor Gary A. Kildall, Code 72Kd Computer Science Group Naval Postgraduate School Monterey, California 93940	1
4.	Instructor Gary Raetz, Ens. USN, Code 72Rr Computer Science Group Naval Postgraduate School Monterey, California 93940	1
5.	Lt. Van D. Underwood, USN Box 241 Osceola, Iowa 50213	1

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) An Investigation of the Parameters for Register Allocation During Compilation		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1974
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Van Douglas Underwood		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1974
		13. NUMBER OF PAGES 31
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

INTERNALLY DISTRIBUTED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

REPORT

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

code improvement	active	basic block
program graph	expressions	common subexpression
register allocation	register	elimination
data item	region	live variable

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Two established methods of code improvement, day '4" and Kildall '7", are reviewed. The problems of optimal register allocation are discussed. A method is presented using Kildall's '7" optimization algorithm for specifying the active data items in a program. Demonstration of particular problems with register allocation are presented. Topics for further consideration in a complete solution are discussed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
(Page 1) S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Thesis [illegible]557
U435 Underwood
c.1 An investigation of the parameters for register allocation during compilation.